Data-Loop-Free Self-Timed Circuit Verification

Cuong Chau, Warren A. Hunt, Jr., Matt Kaufmann
Department of Computer Science
The University of Texas at Austin
Austin, TX, USA
Email: {ckcuong,hunt,kaufmann}@cs.utexas.edu

Marly Roncken, Ivan Sutherland
Maseeh College of Engineering and Computer Science
Portland State University
Portland, OR, USA
Email: mroncken@pdx.edu, ivans@cecs.pdx.edu

Abstract—This paper presents a methodology for formally verifying the functional correctness of self-timed circuits whose data flows are free of feedback loops. In particular, we formalize the relationship between their input and output sequences. We use the DE system, a formal hardware description language built using the ACL2 theorem-proving system, to specify and verify finite-state-machine representations of self-timed circuit designs. We apply a link-joint paradigm to model self-timed circuits as networks of computations that communicate with each other with protocols. Our approach exploits hierarchical reasoning and induction to support scalability. We demonstrate our methodology by modeling and verifying several data-loop-free self-timed circuits.

I. INTRODUCTION

The synchronous or clocked design paradigm dominates digital circuit designs today mostly because of the conceptual simplicity offered by the global clock. The omission of the global clock in self-timed designs offers many potential benefits, such as reduced power consumption, increased throughput, and greater layout freedom. However, the lack of computer-aided design (CAD) tools and verification methods for self-timed design keeps the self-timed paradigm from being widely adopted by industry. The work presented in this paper attempts to improve the verification methodology for self-timed systems. Specifically, we have developed a methodology for formally verifying self-timed systems provided their data flows are free of feedback loops. This is a first step towards our effort to develop a specification and mechanical verification environment for self-timed systems.

Unlike many efforts in validating timing properties of self-timed systems, we are interested in verifying functional properties of these systems. For data-loop-free, self-timed systems, we verify functional correctness in terms of the relationship between their input and output sequences. We view self-timed circuit designs as interconnected networks of finite-state machines. Though correctness of the lower-level circuit implementations may depend on circuit-level timing constraints, we assume that such correctness proofs can be provided separately, as suggested by Park et al. [1].

Our work focuses on developing scalable methods for reasoning about the functional correctness of self-timed systems. Our verification framework appeals to hierarchical reasoning and induction to support scalability. We use the DE system [2], a formal occurrence-oriented hardware description language (HDL) embedded within the ACL2 theorem-proving system [3], [4], to specify and verify self-timed circuit designs. The DE language permits hierarchical definition of finite-state machines (FSMs) in the style of an HDL. The capability of verifying hardware modules hierarchically has made DE a valuable tool for analyzing digital circuit designs, such as clocked microprocessor designs [5], [6]. It also provides a library of verified hardware circuit generators that can be used to synthesize hardware components [5]. Here we show that DE can be adapted for modeling and verifying self-timed systems as well.

Our self-timing model is based on the link-joint model proposed by Roncken et al. [7], a universal model for various self-timed circuit families. In particular, we use DE to model self-timed circuits as networks of communication and computation primitives that conform to the link-joint model. We follow our earlier modeling approach [8], but with an improved verification strategy. Our previous approach suffers from limitations in its verification approach.

- Our previous verification approach specifies explicit interleavings of circuit operations. We proved the functional correctness of the corresponding circuit only if the circuit behavior conformed to the specified interleavings. Our earlier approach fails to prove the completeness of those interleavings; that is, we have no mechanism to assure that all interleavings were considered.

- Our previous verification approach imposes a design restriction preventing a module from communicating with other modules until it became quiescent after finishing all of its internal processing. This restriction would likely reduce the performance of corresponding self-timed circuit implementations.

The approach presented in this paper overcomes these limitations.

- We verify the correctness of circuit behavior without specifying interleavings. Our approach proves that the circuit behavior meets its specification under all possible operation interleavings.

- We develop a new hierarchical verification method that avoids exploring operational interleavings of verified sub-
systems. We show that our methodology scales well even as circuit size increases.

- We avoid the communication restriction imposed by our previous work.

The rest of the paper is organized as follows. Section II overviews the DE system. Section III describes our self-timed circuit modeling and verification approach. Section IV demonstrates our approach by describing our modeling and verification through several case studies. Related work is given in Section V. Concluding remarks and future work appear in Section VI.

II. OVERVIEW OF THE DE SYSTEM

This section briefly describes the DE system, an occurrence-oriented HDL for describing and analyzing hierarchical Mealy machines [2]. The syntax and semantics of the DE HDL are defined as ACL2 functions. DE defines circuit descriptions, called netlists, that can be analyzed with the ACL2 theorem-proving system. A DE netlist defines an ordered list of modules, where each module may include references to other modules declared later in the list or to DE primitives. Each module consists of five ordered entries: a unique module name, input names, output names, internal-state names, and a list of occurrences that reference submodules (which are defined later in the DE netlist) or DE primitives. Each occurrence consists of four ordered entries: a module-unique occurrence name, a list of output names, a reference to a DE primitive or a submodule, and a list of input names. A DE netlist of the half-adder and full-adder shown in Figure 1 appears below. These adders are combinational (state-free) circuits.

(defconst 'netlist* (list 'full-adder (c a b) (sum carry) () ;; () indicates no internal states (t0 (sum1 carry1) half-adder (a b)) (t1 (sum carry2) half-adder (sum1 c)) (t2 (carry) or (carry1 carry2))) 'half-adder (a b) (sum carry) () ;; () indicates no internal states (g0 (sum) xor (a b)) (g1 (carry) and (a b))))

The semantics of the DE language is given by the output "wire" evaluator se and state evaluator de that, given the current inputs and current state for a module, will compute the module’s outputs and next state, respectively. Both evaluators require the name of the module to be evaluated (simulated), the corresponding inputs and state, and a netlist containing the module’s definition, including the definitions of all submodules. Below is an example call to compute the results of simulating the full-adder module using the se evaluator. This evaluator takes four ordered arguments: the name of the module to evaluate, its inputs, its current state, and a netlist containing the module definition. The ACL2 constants t and nil represent Boolean true and false, respectively. The single quotation marks tell the evaluator to take the inputs as given, thus the expression 'full-adder '(t nil t) '() *netlist*)

=' (nil t)

For this work, we use the DE language to represent self-timed circuit descriptions, and we use the DE semantics to calculate outputs and new states. The DE semantics supports hierarchical verification of FSMs described using the DE language. This is critical in verifying the correctness of large circuit descriptions. Centaur Technology uses this approach to verify properties of their x86-compatible microprocessor designs [9]. Our self-timed circuit analysis uses the hierarchical approach provided by the DE system to analyze self-timed circuit descriptions. More specifically, for every DE module we prove two lemmas — a value lemma characterizing a module’s outputs and a state lemma characterizing a module’s next state — and for other than the lowest-level modules, these two lemmas are proved by automatic application of the value and state lemmas of submodules, without diving into the details of the submodules. A purely combinational module lacks an internal state and so requires only a value lemma.

III. MODELING AND VERIFICATION APPROACH

For modeling self-timed circuits, we are using the link-joint model as it is represented by the ACL2 logic. We use the ACL2 theorem prover to certify properties of self-timed circuit designs represented as DE structures.

A. Modeling

We start with a brief description of the link-joint model and its associated terminology. We use DE netlists to represent self-timed circuit designs that satisfy the following criteria.

- State-holding devices update their states based on local signaling without reference to a global clock.
- Links and joints communicate with self-timed protocols.
- Each joint action includes reference to an external go signal that selectively permits the action and we allow distinct values for all go signals.

For establishing local communication protocols, we use ACL2 to model the link-joint model introduced by Roncken et al. [7]. Our rationale for choosing this model is its applicability to all self-timed circuit families. Links are communication
channels in which data are stored along with a validity signal. Joints perform data operations and implement flow control. Joints are the meeting points that coordinate links and share link data. A self-timed system can be viewed as a directed graph with links as edges and joints as nodes: the input or output of a link connects to exactly one joint each, so the link serves as a directed edge between those two joints. Figure 2 shows an example of a simple self-timed communication circuit using the link-joint model. This circuit consists of a joint associated with an input link \( L_0 \) and an output link \( L_1 \). In general, a joint can have several input and output links connected to it, as suggested by Figure 3.

Links receive \textit{fill} or \textit{drain} commands from, and report their full/empty states and data to, their connected joints. A \textit{full} link carries valid data, while an \textit{empty} link holds data that are not yet or no longer valid. When a link receives a fill command at its input end, it changes its state to full. A link will change to the empty state only if it receives a drain command at its output end. We have added a link control primitive to the DE primitive database that models the full/empty state of a link, as illustrated by the lower box in each shown in Figure 2. The two inputs \( F \) and \( D \) of this primitive stand for \textit{fill} and \textit{drain} inputs, respectively. The interested reader may refer to Roncken et al.'s previous work [7] for implementation examples of link control circuitry.

Joints receive the full/empty states of their links and issue the \textit{fill} and \textit{drain} commands when their communication conditions are satisfied. Primitive joints are storage-free and they perform data computation and drain and fill storage links. In general, the control logic of a joint is an AND function of the conditions necessary for it to act. A joint can have multiple such AND functions to guard different actions, which are usually mutually exclusive. To enable a joint action, all input and output links of that action must be full and empty, respectively, as illustrated by the AND gates in Figures 2 and 3. Because of arbitrary delays in wires and gates, joints that are enabled (that is, when at least one action is enabled) may fire in any order. We model this self-timed circuit behavior by associating each joint with a so-called \textit{go} signal as an extra input to the AND function in the control logic of that joint. In case a joint has multiple such AND functions, they may share the same \textit{go} signal when at most one function can fire at a time. The setting of the \textit{go} signal will indicate whether the corresponding joint will fire when it is enabled [10]. In our framework, when applying the \textit{de} function that computes the next state of a self-timed circuit, only enabled joints with enabled \textit{go} signals will fire. When a joint action fires, the following three tasks will execute in parallel:  

- using data from input links, compute and transfer results to output links;
- \textit{fill} the output links, leaving them full; and
- \textit{drain} the input links, leaving them empty.

Below is a DE description of the self-timed module shown in Figure 2, where \( D_0 \) and \( D_1 \) are latches. The Combinational Logic oval represents the data computation of the joint which is simply a storage-free buffer in this DE description. This diagram shows four state-holding devices that work together in pairs: data latch \( D_0 \) \((d0\text{ below})\) and its associated full/empty flag \( S_0 \) \((s0\text{ below})\), and data latch \( D_1 \) \((d1\text{ below})\) and its associated full/empty flag \( S_1 \) \((s1\text{ below})\).

```plaintext
\{ (link-joint
  (fireA fireC d0-in go)
  (s0-status s1-status d1-out))
(s0 d0 s1 d1) ;; Internal states

;; Link L0
(s0 (s0-status) link-cntl (fireA fireB))
(d0 (d0-out d0-out-) latch (fireA d0-in))

;; Link L1
(s1 (s1-status) link-cntl (fireB fireC))
(d1 (d1-out d1-out-) latch (fireB d1-in))

;; Joint
  (fireB)
  joint-cntl
  (s0-status s1-status go)
  (h (d1-in) buffer (d0-out)))
\}
```

### B. Verification

We develop a methodology for verifying the functional correctness of self-timed systems in terms of the relationship between their input and output sequences. Our present methodology handles self-timed circuits without feedback loops in their data flows. Our key proof strategy is based on the single-step, state-transition formalization of a self-timed model at the “functional” level, as described below. We call this formalization the \textit{single-step-update} property.

\[
\text{extract}(\text{step}(\text{input}, \text{st})) = \text{extracted-step}(\text{input}, \text{st})
\]

(1)

Given the above formalization, the relationship between input and output sequences of a self-timed circuit is proved via induction. In equation (1), function \textit{step} returns the circuit’s next state, as produced by the \textit{de} function; function \textit{extract(st)} returns a sequence of values computed from state \textit{st} that

\footnote{The work done by Park et al. [1] used ARCtimer to generate and validate timing constraints in joints. Their framework added sufficient delay to the control logic of each joint to guarantee that the fire pulse is enabled for long enough for the three actions to execute properly when the joint acts. Our work assumes that we have a valid circuit that satisfies necessary circuit-level timing constraints, as might be guaranteed by ARCtimer.}
may ultimately appear in the output sequence; and function \textit{extracted-step} specifies the extracted values from the next state in terms of the \textit{extract} function. An important property of \textit{extracted-step} is that its definition avoids exploring the possible interleavings; it depends only on the communication signals at the module’s input and output ports. In the case studies (Section IV), we will discuss the use of \textit{extract} and \textit{extracted-step} in more detail.

Our framework exploits a hierarchical verification approach to formalizing single transitions of circuit behavior at the link-joint level (as emulated by the \textit{se} and \textit{die} functions). This is the same hierarchical verification approach we previously reported [8]. We have also developed a hierarchical verification method for establishing the single-step-update property. We prove the single-step-update property of a module hierarchically by using the single-step-update properties of its submodules, thus avoiding exploration of the operational interleavings of submodules. Much of this proof process is stylized and automatic.

The verification process at the module level requires us to show how to interconnect several self-timed blocks to provide provably correct, higher level functions. Self-timed modules can be abstracted as “complex” links or “complex” joints. A module is called a complex link if only links appear at its input and output ports (Figure 4). Similarly, a module is called a complex joint if only links appear at its input and output ports (Figure 5). A complex joint can replace any other joint in the system that has the same configuration of inputs and outputs, without violating the link-joint topology: links are connected via joints, and joints are connected via links. It is typical that self-timed modules receive data in one end and send data out the other end, via different links, using separate input and output communication signals.

We generally model self-timed modules as complex joints, as illustrated in Section IV. However, we also demonstrate the potential advantage of complex links for significantly improving the verification time of a circuit; see Section IV-D.

IV. CASE STUDIES

In this section we demonstrate our framework through case studies of data-loop-free self-timed circuits 2.

A. Example 1

Our first example is a queue of length three called \textit{Q3}. This circuit contains three links and four joints that pass data items from its input to its output (Figure 7). Let \textit{in-act} denote the fire signal from the AND gate (not illustrated) in the control logic of joint \textit{in} and let \textit{out-act} denote the fire signal from the AND gate (not illustrated) in the control logic of joint \textit{out}. Module \textit{Q3} accepts a new data item each time the \textit{in-act} signal fires. We define \textit{in-seq}, the accepted input sequence, as the sequence of data items that have passed joint \textit{in}. Similarly, we define \textit{out-seq}, the valid output sequence, as the sequence of data items that have passed through joint \textit{out}. \textit{Q3} will deliver a valid data item each time joint \textit{out} acts. Note that \textit{in-act} cannot fire when \textit{L0} is full, and \textit{out-act} cannot fire when \textit{L2} is empty.

The extraction function \texttt{q3$\text{extract(}\text{st}\text{)}$} simply extracts valid data from \textit{Q3}; in particular, it extracts data from links that are full at state \textit{st}, preserving the queue order. For example, suppose links \textit{L0} and \textit{L2} are full, and link \textit{L1} is empty in state \textit{st}; then \texttt{q3$\text{extract(}\text{st}\text{)}$} = \{d0, d2\}, where \textit{di} is the data item of link \textit{Li}. We define the extracted next-state function \texttt{q3$\text{extracted-step}$} to extract valid data at the next state of \textit{Q3}; this is defined in terms of \texttt{q3$\text{extract}$}, as shown in Figure 6. Note that the parameter \textit{input} we mention throughout our case studies consists of both input data and input control signals, including \textit{go} signals for every joint. We write \textit{input.data} to denote the data part of the input. Function \texttt{q3$\text{extracted-step}$} avoids considering interleavings of \textit{Q3}; it considers only the values of the \textit{in-act} and \textit{out-act} signals at \textit{Q3}’s input and output ports respectively, thus reducing the

2The source code for this work is available at https://github.com/acl2/acl2/tree/master/books/projects/async/fifo
if

3

nil

q

nil

5

5

Q

st

delivers all data from the initial state and is empty, the link providing the remove

that are accepted by 3,

L

Q

= 

when the initial

− 

= 

by proving that

= 

++

returns

8

except for its last element.

in

∧ 

[1

4

++

5
]= 

[1

3

∧ 

∧ 

3

lin

is

++

was

8

is

out

[8

will deliver the

= 

otherwise

4

and

++

(3

contain only data (not control). Moreover, data

→

returns the list generated by mapping over the

Q

is the operation that zips together two lists, e.g.,

A

→

returns

data

→

Q

q

⊗

is the number

B

[5

and

over a data sequence is

c

that contains

q

++

and

v

NULL

= 

( 

and

: =

[ 

a

b

a

as

in

Q

is identical to the concatenation of the input sequence and the

valid data in the initial state. Let us represent invalid data as

Q

= 3

where

++

is the concatenation operation and remove-last(l) returns list l except for its last element.

Fig. 6. Definition of q3$extracted-step

Fig. 7. Data flow of Q3: a queue of three links

complexity of extracting valid data from Q3’s next state to

the four cases shown in Figure 6.

We verify the functional correctness of Q3 by proving that

after an n-step execution from its initial state, the concatenation

of the valid data in the final state and the output sequence is identical to the concatenation of the input sequence and the valid data in the initial state. Let us represent invalid data as

Q

= 3

Given that the input sequence consumed by Q3 is [1, 4, 3],

and the initial content of Q3 was [8, _ — 5], Q3 will deliver the valid data from its initial state first then followed by the input sequence. If Q3 delivers all data from the initial state and the input sequence, the output sequence must be [1, 4, 3, 8, 5].

Specifically, the output sequence is the concatenation of the input sequence and the valid data of the initial state. However, suppose at some time the content of Q3’s state is [1, _ — 3, _ — 5], then the output sequence must be [4, 3, 8, 5]. In this case, the following equation states the relationship between the input and output sequences of Q3: [1] ++ [4, 3, 8, 5] = [1, 4, 3] ++ [8, 5]. This relation is formalized as follows,

q3$extract(q3$run(input-list, st, n)) ++ out-seq =
in-seq ++ q3$extract(st) ⑵

where q3$run(input-list, st, n) :=

if (n ≤ 0)

then st

else q3$run(rest(input-list),

q3$step(first(input-list), st),

n − 1)

It trivially follows that out-seq = in-seq when the initial

and final states contain no valid data. Note that in-seq and

out-seq contain only data (not control). Moreover, in-seq is

extracted from the inputs in input-list that are accepted by

Q3, specifically when: L0 is empty, the link providing the

input data is full, and the corresponding go signal is active.

Our ACL2 proof of (2) uses induction and the following

single-step-update property, which is an instance of (1), as a

supporting lemma.

q3$extract(q3$step(input, st)) = q3$extracted-step(input, st)

B. Example 2

The next example illustrates how we apply hierarchical

reasoning to verify a circuit C that contains Q2 and Q3 as

submodules (Figure 9). In terms of input-output relationship,

C simply performs the bitwise OR operation on paired input
data. Joint in splits the input data items into two, equal-sized

fields, a and b. The operation of C over a data sequence is

specified as follows,

c$op-seq(seq) :=

if (seq = NULL) then [] / / an empty list

else let in := first(seq)

   return [v-or(in.a, in.b)] ++ c$op-seq(rest(seq))

where v-or(a, b) performs the bitwise OR operation over

fields a and b. We then define an extraction function that

computes the future output sequence from the current state,

st, as described below,

c$extract(st) :=
c$op-seq

(data([st.A0]) ++ q2$extract(st.Q2) ++ data([st.A1]))

⊗

(data([st.B0]) ++ q3$extract(st.Q3) ++ data([st.B1]))

where ⊗ is the operation that zips together two lists, e.g.,

[1, 3, 5] ⊗ [2, 4, 6] = [[1, 2], [3, 4], [5, 6]]. The projection function data(l) returns the list generated by mapping over the

links in l, collecting the data item of each full link and ignoring

each empty link. A key invariant for the state of C is that the

number of valid data of queue (A0 → Q2 → A1) equals

the number of valid data of queue (B0 → Q3 → B1). We

formalize this invariant as follows (where len(l) is the number

of elements in list l).

c$inv(st) :=

(len(data([st.A0]) ++ q2$extract(st.Q2) ++ data([st.A1]))

= len(data([st.B0]) ++ q3$extract(st.Q3) ++ data([st.B1])))
The relationship between input and output sequences is nil and (4) avoids considering the internal details of $Q$. In other words, we employ a hierarchical reasoning, illustrated in Figure 10. The branch joint takes data alternately from links holds, assuming $c^\text{++}$, and delivers them as outputs. Links $I_0$ and $O_0$ hold Boolean values that retain the alternation state. When the branch joint fires, it fills either link $L_0$ or link $L_1$ according to the input alternation state (false or true, respectively). Likewise, the merge joint will drain either $L_0$ or $L_1$ when it fires, according to the output alternation state. In addition, the branch joint delivers the negated value of $I_0$ to $I_1$. This new alternation value returns to the branch joint via $I_0$. The same mechanism applies in the merge joint. An interesting property of WW is that its functionality is equivalent to $Q_2$, but potentially has higher performance because of its shorter latency.

Given that the full/empty states of $I_0$ and $I_1$ must differ, we define the select signal of the branch joint as follows.

$$\text{branch-select}(st) :=$$

- if full(st.$I_0$.status)
  - then st.$I_0$.data
  - else st.$I_1$.data

Similarly, the definition of the select signal of the merge joint is given below.

$$\text{merge-select}(st) :=$$

- if full(st.$O_0$.status)
  - then st.$O_0$.data
  - else st.$O_1$.data

We verify the correctness of WW by proving the same property we used to specify $Q_2$: the concatenation of valid internal data in the final state with the output sequence is identical to the concatenation of the input sequence with the initial valid data.

$$\text{ww}$extract(ww$run(input-list, st, n)) \oplus out-seq = in-seq \oplus \text{ww}$extract(st) \quad (5)$$
The extraction function for WW is defined as follows.

\[
ww\text-extract(st) := \\
\text{if } (\text{merge-select}(st) = t) \\
\text{then data([st.L_0, st.L_1]))} \\
\text{else data([st.L_1, st.L_0]))}
\]

Our proof of (5) requires the initial state st to satisfy the following condition.

\[
ww\text-inv(st) := \\
\text{st.L_0.status \neq st.I_1.status} \land \\
\text{st.O_0.status \neq st.O_1.status} \land \\
\left( \text{st.L_0.status = st.L_1.status} \land \\
\text{branch-select(st) = merge-select(st))} \lor \\
\text{(full(st.L_0.status) \land empty(st.L_1.status))} \land \\
\text{branch-select(st) = t} \land \text{merge-select(st) = nil} \lor \\
\text{(empty(st.L_0.status) \land full(st.L_1.status))} \land \\
\text{branch-select(st) = nil} \land \text{merge-select(st) = t)}
\]

Assuming the current state st satisfies the \(ww\text-inv\) condition, we prove the single-step-update property of WW as an auxiliary lemma for proving (5). In addition, in order to prove (5) by induction, we also prove that \(ww\text-inv\) is an invariant.

\section*{D. Example 4}

In this example, we demonstrate the benefit of specifying a module as a complex link. This approach aids our proof of both a self-timed system’s invariant as well as its single-step-update property. First, let us consider an extended version of the wig-wag circuit WW in which links \(L_0\) and \(L_1\) are replaced by the queues \((A_0 \rightarrow Q_2 \rightarrow A_1)\) and \((B_0 \rightarrow Q_3 \rightarrow B_1)\), as shown in (Figure 11). We call this circuit round-robin \(RR1\). Recall that \(Q_2\) and \(Q_3\) are complex joints (Figures 5 and 7). We apply the same verification procedure as described in previous examples to verify the relationship between input and output sequences.

The verification time of \(RR1\) is more than 23.5 minutes, while verifying \(WW\) takes only 9 seconds on a contemporary laptop. Why? There are many case splits required to prove the invariant as well as the single-step-update property for \(RR1\). It takes 3.5 minutes to prove the invariant and nearly 20 minutes to prove the single-step-update property. Most case splits arise from considering the full/empty states of four links \(A_0, A_1, B_0,\) and \(B_1\) along with the case splits in the correlation between the numbers of valid data items in \(Q_2\) and \(Q_3\).

To reduce the number of case splits in this problem we abstract two queues \((A_0 \rightarrow Q_2 \rightarrow A_1)\) and \((B_0 \rightarrow Q_3 \rightarrow B_1)\) as two complex links. We call these two links \(Q_4\) and \(Q_5\); for \(Q_4\) see Figure 12. We follow the same procedure of formalizing the relationship between input and output sequences for \(Q_4\) and \(Q_5\), as described in previous examples. The verification times of \(Q_4\) and \(Q_5\) are 4 and 7 seconds respectively. By using \(Q_4\) and \(Q_5\) as submodules we construct an alternative round-robin circuit, which we call \(RR2\), not illustrated. The verification time of \(RR2\) is a mere 14 seconds, which shows the benefit of using a hierarchical verification approach with complex links.

To demonstrate the scalability of our approach, we specify and verify a larger round-robin circuit, which we call \(RR3\). \(RR3\) replaces \(Q_4\) and \(Q_5\) with longer queues \(Q_8\) and \(Q_{10}'\), respectively. \(Q_8\) is a complex link representing a queue of eight links and is constructed by concatenating two instances of \(Q_4\) via a buffer joint. Similarly, \(Q_{10}'\) is constructed by concatenating two instances of \(Q_5\). Our proof of \(RR3\) is exactly the same as that of \(RR2\) and its verification time is also 14 seconds. The verification times for \(Q_8\) and \(Q_{10}'\) are 3 seconds each. Figure 13 reports the verification times of the self-timed circuits discussed in our case studies.

\section*{V. RELATED WORK}

Many research efforts in self-timed circuit verification have focused on verifying properties of circuits by applying timing verification techniques [11], [12], [13], [14], [1]. Park et al. [1] presented their framework, called ARCtimer, for modeling, generating, verifying, and enforcing timing constraints for individual delay-insensitive handshake components. ARCtimer uses the general-purpose model checker NuSMV to perform timing verification of handshake components. Their goal was to verify that the network of logic gates and wires, with their associated delays, meets its communication protocol
specification. In contrast, our goal is to verify that the network of components and their protocols meets its functional specification, while ignoring circuit-level timing constraints that can be investigated by tools like ARCTimer.

Our earlier work [8] demonstrated the functional correctness of a self-timed circuit containing a data feedback loop, namely, a 32-bit self-timed serial adder. That approach requires the explicit specification of all possible interleavings of circuit operations by conditioning the values of go signals. However, we failed to prove that those interleavings actually cover all possible execution paths of the circuit. Our present approach is more general in the sense that it avoids imposing any conditions on the values of go signals. As a consequence, our present framework allows stuttering in circuit executions as simulated by the DE evaluator. This framework also incorporates a hierarchical reasoning approach that avoids the need to consider interleavings within verified submodules.

Using ACL2, Verbeek and Schmalz [15] formalized and verified blocking (not transmitting data) and idle (not receiving data) conditions about delay-insensitive primitives from the Click circuit library. These conditions were then used to derive SAT/SMT instances to check deadlock freedom in the self-timed circuits developed. While our approach also uses ACL2 to model and verify self-timed circuits, we verify the functional correctness of self-timed circuit models.

The idea of employing a hierarchical method in self-timed circuit verification was also explored by Clarke and Mishra [16], in their attempt to verify automatically safety and liveness properties of a self-timed FIFO queue element using model checking. Nevertheless, their approach imposed an unrealistic assumption on self-timed circuits that each gate had a unit delay. Our approach, on the other hand, avoids imposing any restrictions on gate delays.

VI. Conclusion and Future Work

In this paper we show that our methodology applies to verifying the functional correctness of data-loop-free self-timed circuits modeled with the link-joint paradigm. For each data-loop-free self-timed module, we first prove its single-step-update property and, when needed, an invariant by using model checking. Nevertheless, their approach imposed an unrealistic assumption on self-timed circuits that each gate had a unit delay. Our approach, on the other hand, avoids imposing any restrictions on gate delays.

The approach of employing a hierarchical method in self-timed circuit verification was also explored by Clarke and Mishra [16], in their attempt to verify automatically safety and liveness properties of a self-timed FIFO queue element using model checking. Nevertheless, their approach imposed an unrealistic assumption on self-timed circuits that each gate had a unit delay. Our approach, on the other hand, avoids imposing any restrictions on gate delays.

In this paper we show that our methodology applies to verifying the functional correctness of data-loop-free self-timed circuits modeled with the link-joint paradigm. For each data-loop-free self-timed module, we first prove its single-step-update property and, when needed, an invariant by using the single-step-update properties of its submodules. The approach is hierarchical: it ignores the internal details of referenced submodules. We then prove the functional correctness of that self-timed module in terms of the relationship between its input and output sequences using induction and its single-step-update property. Our verification approach avoids specifying interleavings of circuit operations, and hence we verify the correctness of circuit behavior under all possible internal interleavings. As far as we know, this is the first complete hierarchical functional verification approach for data-loop-free self-timed circuits designed at the protocol link-joint level.

In the future, we plan to explore strategies for verifying the functional correctness of iterative self-timed circuits. Such circuits must contain feedback loops in their data flows. Like our present approach described here, we aim to verify iterative self-timed circuits without imposing interleavings (with go-signal values). Coupled with our history in the verification of arithmetic-and-logic-intensive circuits, we hope to verify self-timed implementations of large systems.

Acknowledgments

We thank Anna Slobodova for her useful comments and corrections on this paper. We also thank the reviewers for useful feedback. This material is based upon work supported by DARPA under Contract No. FA8650-17-1-7704.

References